Props

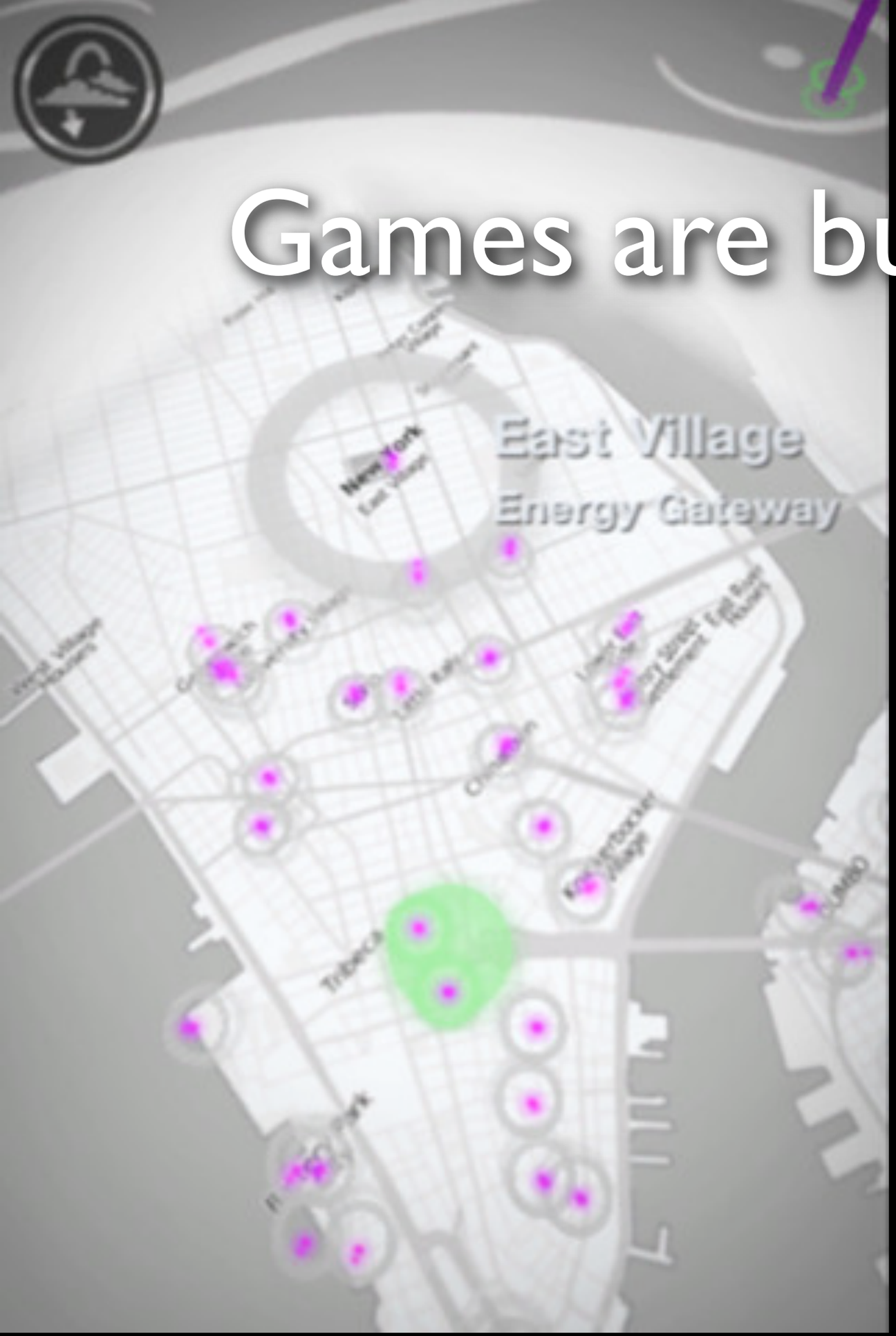
A DSL for Manipulating JSON-like Structures in Erlang

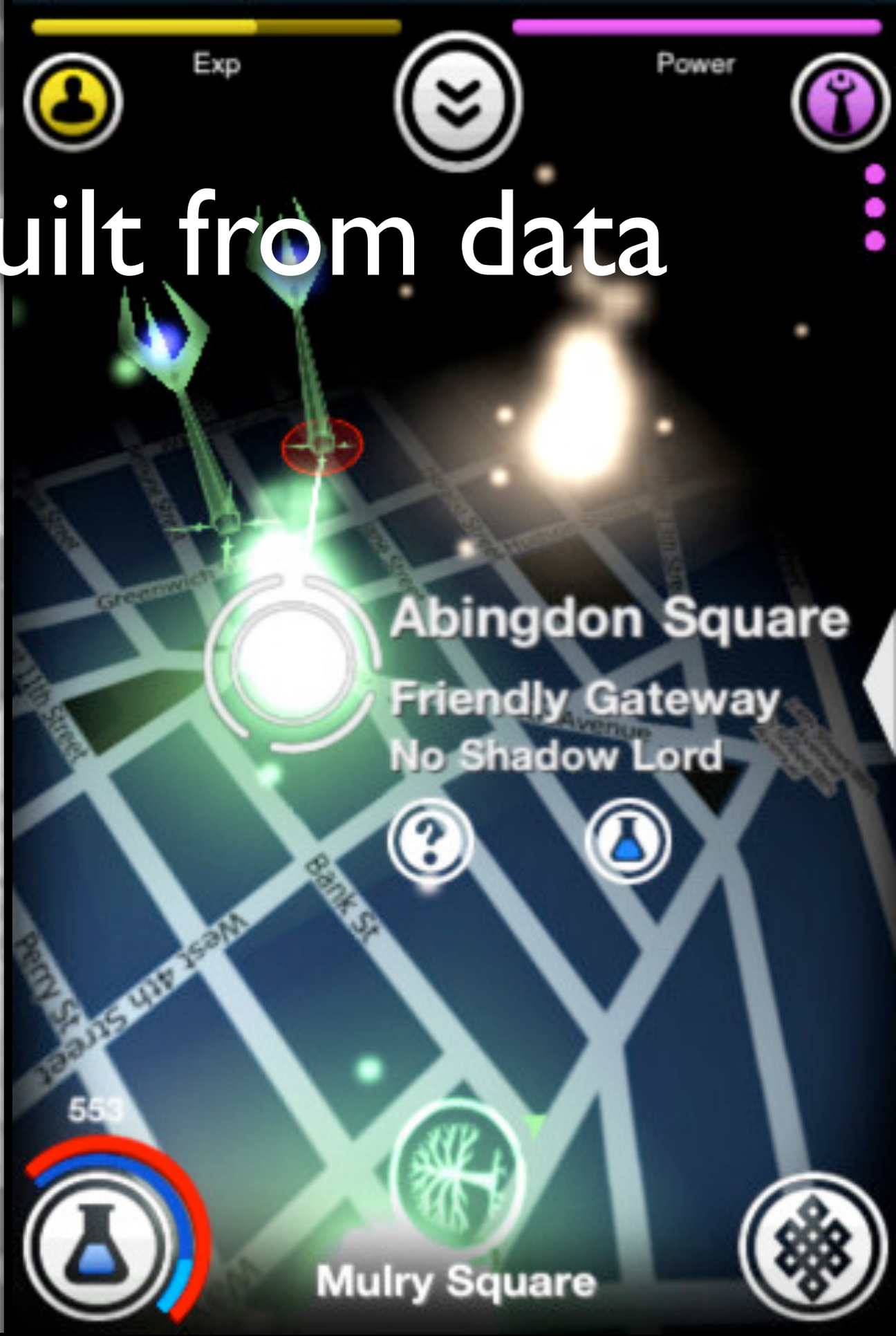
http://github.com/greyarea/props

Jack Moffitt
jack@metajack.im
@metajack
http://metajack.im

Motivation

Games are built from data





Entity systems

Protocols and APIs are built on ata

How to represent data in Erlang?

```
7> NestedBot = #robot{details=#robot{name="erNest"}}.
#robot{name = undefined,type = industrial,
       hobbies = undefined,
       details = #robot{name = "erNest",type = industrial,
                        hobbies = undefined,details = []}}
8> (NestedBot#robot.details)#robot.name.
"erNest"
```

http://learnyousomeerlang.com/a-short-visit-to-common-data-structures

JSON

```
%% strings - "foobar"
<<"foobar">>
```

```
%% empty object - {}
{[]}
```

```
%% array - [1, 2, {}]
[1, 2, {[]}]
```

```
%% nested objects
%% {"foo": {"bar": 1, "baz": [2]}}
{[{<<"foo">>, {[{<<"bar">>, 1},
                {<<"baz">>, [2]}]}}]}
```

Goals

Access and set deeply nested values in structures

Make broad edits easily

Comparison tools

Pretty printing, fold, format conversion, ...

Path DSL

Keys:

foo

Nested keys

foo.bar

Array indices

foo.bar[2].baz

Specified as atoms:

foo.bar.baz

'foo[2].bar'

Specified as strings:

"foo.bar"

"foo[2].bar"

Need to write a parser

Neotoma

PEG parser

Ordered choice:

CFG:

foo ::= bar | baz

PEG:

foo ::= bar / baz

No tokenization pass needed

Downsides?

Grammar and semantic actions

```
var <- [_a-zA-Z0-9] [:\s_a-zA-Z0-9]* `
list_to_binary(Node)`;
```

```
var <- [_a-zA-Z0-9] [:\s_a-zA-Z0-9]* `
list_to_binary(Node)`;
```

```
var <- [_a-zA-Z0-9] [:\s_a-zA-Z0-9]* `
list_to_binary(Node)`;
```

```
var <- [_a-zA-Z0-9] [:\s_a-zA-Z0-9]* `
list_to_binary(Node)`;
```

```
%% NOTE: Node = [<<"foo">>]
```

```
int <- [0-9]+ `
  list_to_integer(
binary_to_list(
  list_to_binary(Node)))`;
```

Result of parse:

[{prop, <<"foo">>}, {index, 2}, ...]

```
path <- var ('[' int ']')? ('.' path)? `
  case Node of
      [Var, [], []] ->
          [{prop, Var}];
      [Var, [_, I, _], []] ->
          [{prop, Var}, {index, I}];
      [Var, [], [_, Rest]] ->
          [{prop, Var}] ++ Rest;
      [Var, [_, I, _], [_, Rest]] ->
          [{prop, Var}, {index, I}] ++ Rest
end`;
```

```
path <- var ('[' int ']')? ('.' path)? `
   case Node of
        [Var, [], []] ->
            [{prop, Var}];
        [Var, [_, I, _], []] ->
            [{prop, Var}, {index, I}];
        [Var, [], [_, Rest]] ->
            [{prop, Var}] ++ Rest;
        [Var, [_, I, _], [_, Rest]] ->
            [{prop, Var}, {index, I}] ++ Rest
end`;
```

```
path <- var ('[' int ']')? ('.' path)? `
    case Node of
        [Var, [], []] ->
            [{prop, Var}];
        [Var, [_, I, _], []] ->
            [{prop, Var}, {index, I}];
        [Var, [], [_, Rest]] ->
            [{prop, Var}] ++ Rest;
        [Var, [_, I, _], [_, Rest]] ->
            [{prop, Var}, {index, I}] ++ Rest
end`;
```

```
path <- var ('[' int ']')? ('.' path)? `
   case Node of
        [Var, [], []] ->
            [{prop, Var}];
        [Var, [_, I, _], []] ->
            [{prop, Var}, {index, I}];
        [Var, [], [_, Rest]] ->
            [{prop, Var}] ++ Rest;
        [Var, [_, I, _], [_, Rest]] ->
            [{prop, Var}, {index, I}] ++ Rest
end`;
```

```
path <- var ('[' int ']')? ('.' path)? `
   case Node of
        [Var, [], []] ->
            [{prop, Var}];
        [Var, [_, I, _], []] ->
            [{prop, Var}, {index, I}];
        [Var, [], [_, Rest]] ->
            [{prop, Var}] ++ Rest;
        [Var, [_, I, _], [_, Rest]] ->
            [{prop, Var}, {index, I}] ++ Rest
end`;
```

```
path <- var ('[' int ']')? ('.' path)? `
   case Node of
        [Var, [], []] ->
            [{prop, Var}];
        [Var, [_, I, _], []] ->
            [{prop, Var}, {index, I}];
        [Var, [], [_, Rest]] ->
            [{prop, Var}] ++ Rest;
        [Var, [_, I, _], [_, Rest]] ->
            [{prop, Var}, {index, I}] ++ Rest
end`;
```

```
path <- var ('[' int ']')? ('.' path)? `
  case Node of
      [Var, [], []] ->
          [{prop, Var}];
      [Var, [_, I, _], []] ->
          [{prop, Var}, {index, I}];
      [Var, [], [_, Rest]] ->
          [{prop, Var}] ++ Rest;
      [Var, [_, I, _], [_, Rest]] ->
          [{prop, Var}, {index, I}] ++ Rest
end`;
```

```
path <- var ('[' int ']')? ('.' path)? `
  case Node of
      [Var, [], []] ->
          [{prop, Var}];
      [Var, [_, I, _], []] ->
          [{prop, Var}, {index, I}];
      [Var, [], [_, Rest]] ->
          [{prop, Var}] ++ Rest;
      [Var, [_, I, _], [_, Rest]] ->
          [{prop, Var}, {index, I}] ++ Rest
end`;
```

```
1> props_path_parser:parse("foo[2]").
[{prop,<<"foo">>},{index,2}]
```

```
2> props_path_parser:parse("foo.bar.baz").
[{prop,<<"foo">>},
 {prop,<<"bar">>},
 {prop,<<"baz">>}]
```

Easy!

Props

Example =

```
{[{<<"foo">>, 1},
  {<<"bar">>, {[{<<"baz">>, 2}]}}]}.
```

Basics

new/0

```
props:new().
```

```
{[]}
```

get/2, get/3

```
props:get(foo, Example). % 1

props:get(foo.bar, Example, not_found). % 2
```

```
set/3
props:set(foo, 1, props:new()).

% {[{<<"foo">>, 1}]}
```

set/2

```
props:set([{foo, 1}, {bar, 2}], props:new()).
% {[{<<"foo">>, 1},
    {<<"bar">>, 2}]}
```

set/2

```
props:set(foo, 1).

% {[{<<"foo">>, 1}]}
```

set/1

```
props:set([{foo, 1}, {bar, 2}]).
% {[{<<"foo">>, 1},
    {<<"bar">>, 2}]}
```

What about?

```
props:set(foo.bar.baz, 2).
```

```
props:set('foo.bar[1]', 2).
```

What about?

```
props:set(foo.bar.baz, 2).
```

```
% {[{<<"foo">>, {[{<<"bar">>, {[{<<"baz">>, 2}]}}]}}]}
```

```
props:set('foo.bar[1]', 2).
```

```
% {[{<<"foo">>, {[{<<"bar">>, [2]}]}}]}
```

Functional things

take/2, drop/2

```
take([bar], Example).
% {[{<<"bar">>, {[{<<"baz">>, 2}]}}]}.

props:drop([foo, bar.baz], Example).
% {[{<<"bar">>, {[]}}]}
```

fold/3

```
props:fold(fun({K, V}, Acc) ->
               Acc + 1
           end, 0, Example).
```

```
% 2
```

Serialization & Deserialization

```
to_proplist/1

props:to_proplist(Example).
[{<<"foo">>, 1},
 {<<"bar">>, [{<<"baz">>, 2}]}]
```

to_pretty/1

```
io:format("~s~n", [props:to_pretty(Example)]).

            {
                foo: 1,
                bar: {
                    baz: 2
                }
            }
```

to_string/1,
from_mochijson2/1,
from_proplist/1

Miscellaneous

merge/2

```
props:merge(Props1, Props2).
```

```
keys/1
  props:keys(Examples).

% [<<"foo">>, <<"bar">>]
```

nested_keys/1

```
props:nested_keys(Example).

% [<<"foo">>,<<"bar.baz">>]
```

diff/2

```
props:diff(Example, props:set(foo, 1)).

% {'bar.baz',{2,undefined}}]
```

select_matches/2, delete_matches/2

```
props:select_matches([Example, Example], props:set(foo, 1)).

% [Example, Example]

props:delete_matches([props:new(), Example], props:set(foo, 1)).

% [{[]}]
```

Other DSLs

Event matching

Examples:

```
attack:weapon=gun
spell
get_item:weight
get_item:!weight
spell:type!=[heal,warp]
```

Animation playlists

Examples:

```
multi_attack(PlayerId, Targets) ->
    frl_playlist:make(
  [[[run(PlayerId, TargetId),
     {hit(PlayerId, TargetId), damage(TargetId, DmgAmount)}]
    || {TargetId, DmgAmount} <- Targets],
   run(PlayerId, PlayerId)]).
```

http://github.com/greyarea/props

Questions?

Jack Moffitt
jack@metajack.im
http://metajack.im
@metajack